Natural Language Understanding, Generation, and Machine Translation (2024–25)

2025 edition: Alexandra Birch, Gustavo Cilleruelo
Past editions: Jonas Waldendorf, Christos Baziotis, Arturo Oncevay, Adam Lopez, Mirella Lapata, and Frank Keller
School of Informatics, University of Edinburgh

NLU+ Coursework 1: Recurrent Neural Networks

This assignment is due on Friday, the 14th of February 2025, at 12pm, UTC.

Deadline for choosing partners: Tuesday, the 21th of January 2025 at 12pm, UTC.

Executive Summary. This first part of this coursework will walk you through the implementation of a few critical parts of a recurrent neural network and the backpropagation algorithm, which are foundational components of the modern NLP toolkit. You will not need to derive anything mathematically—the maths will be given to you. This is similar to a lot of practical work in research and industry—it is *much* more common to implement and test an existing model from a specification than to derive a new one from first principles. In fact, what we ask you to implement is relatively low level, compared to what you can do with modern machine learning libraries. Nevertheless, the implementation-from-specification pattern is similar, and we hope that revealing more of the mathematical details will help demystify the underlying algorithms for you, and give you some degree of comfort with them. If you are very interested in deriving algorithms like this, we encourage you to dig further into the mathematics, but nothing in the coursework requires this.

The next parts of the coursework ask you to experiment with training regimes, and to adapt the model to an interesting psycholinguistic task that tests the model's behaviour on a phenomenon that humans process effortlessly—number agreement between subject and predicate in English. We strongly advise that you read the accompanying research paper from which the coursework dataset orginates (Linzen *et al.*, 2016). Using this setup you will compare the ability of recurrent neural networks and gated recurrent units to learn long range dependencies.

Finally, there is an open-ended final question for especially keen students. **You do not need to answer the final question to receive a good mark**, and should only attempt

it if you are confident in your solutions to the rest of the coursework and you have sufficient time remaining.

Submission Deadline and Pacing. The coursework is due on Friday, the 14th of February at 12pm (UTC), in week 5. If you want to work with a partner, you have an earlier deadline to relate this to us, on Tuesday, the 21th of January at 12pm (UTC) i.e. three weeks before the coursework due date.

There are four questions. Simply answering the first three will earn you a good mark, and they can be easily done in the available time. This is an empirical observation from previous offerings of this course, which included a similar (though not identical!) coursework. The optional open-ended part of the coursework is Question 4. If you want to attempt it, you will need to finish the first four questions a week early. This is a more ambitious schedule, so plan accordingly.

Pair work policy. You are *strongly* encouraged to work with a partner on this assignment.[1] When you work with another person, you learn more, because you need to explain things to each other as you pool your collective expertise to solve problems. Explaining something to another person helps you debug your own thinking, and their questions help you overcome your own blind spots—something you cannot do on your own by staring at maths, code, or data. For this reason, it is best to seek partners with complementary skills to your own. You may not work in teams of three or more.

If you work with a partner, only one of you should submit your completed work. But I need to know that the submission represents the work of two people, and I need to know that reliably in advance in order to estimate marking hours. So, **you must do the following to ensure that both of you receive credit:**

1. Find a partner and confirm which one of you will submit the partnering form. If this is you, **confirm you have correctly recorded your partner's student number (UUN)**.

2. Submit your names and UUNs using the following form:
   `Link to form`
   This form is only accessible when you are logged into your University email account, so you may have to enter your password.
   With this form, you can also notify us whether you want to be randomly paired with a partner.

You must do this by **Tuesday, the 21th of January at 12pm (UTC)**. You may not change partners after you have done this, so take this commitment seriously. If I do

[1] You will follow a similar partnering procedure for Coursework 2.

not receive a submission from you or your partner by the deadline, I will assume that you are working alone. Either way, I will confirm your choice with you shortly after the deadline to avoid later confusion. **I advise you to choose your partner now and get to work.** Use EdStem to find a partner by posting and tagging your post with the "Find Partner" tag. We can also assign you a partner if you need one. You can select that option in the partnering form.

Submission. Your solution should be delivered in two parts and uploaded to Blackboard Learn. **Do not include your name or your partner's name in either the code or the writeup.** The coursework will be marked anonymously since this has been empirically shown to reduce bias. (I anonymize the filenames before marking.)

For your writeup:

- Write up your answers in a file titled `<UUN>.pdf`. For example, if your UUN is `S123456`, your corresponding PDF should be named `S123456.pdf`.
- The answers should be clearly numbered and can contain text, diagrams, graphs, formulas, as appropriate. Do not repeat the question text. If you are not comfortable with writing math on Latex/Word you are allowed to include scanned handwritten answers in your submitted pdf. You will lose marks if your handwritten answers are illegible.
- Lean Ultra, Assessment, select Gradescope CW1 Report. Upload your `<UUN>.pdf` to this assignment, and use the submission title `<UUN>`. So, for above example, you should enter the submission title `S123456`.
- Please make sure you have submitted the right file. We cannot make concessions for students who turn in incomplete or incorrect files by accident.

For your code and parameter files:

- Compress your code for `rnn.py`, `gru.py` and `runner.py` as well as your saved parameters `rnn.U.npy`, `rnn.V.npy`, and `rnn.W.npy` into a ZIP file named `<UUN>.zip`. For example, if your UUN is `S123456`, your corresponding ZIP should be named `S123456.zip`.
- On Learn Ultra, Assessment select Gradescope CW2 Code. Upload your `<UUN>.zip` to this assignment, and use the submission title `<UUN>`. So, for above example, you should enter the submission title `S123456`.

Good Scholarly Practice Please remember the University requirement as regards all assessed work for credit. Details and advice about this can be found at:

`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct` and links from there. Note that you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put your work in a public repository then you must restrict access only to yourself and your partner. For your writeup, and particularly on the final question, you should pay close attention to the guidance on plagiarism. In short: the litmus test for plagiarism is not the Turnitin check—that is simply an automated assistant. If you have borrowed or lightly edited someone else's words, you have plagiarised. Write your report in your own words. We are not marking for eloquence—as long as we can clearly understand what you did, that is fine.

School policy requires you to complete coursework yourself, using your own words, code, figures, etc. and to acknowledge any sources of text, code, figures etc. that are not your own. This policy includes the use of intelligent writing assistants such as ChatGPT. Using such an assistant without acknowledgement is a form of academic misconduct.

Assignment Data Files for this assignment are available on the course page in Learn.

Python Virtual Environment For this assignment you will be using Python along with a few open-source packages. These packages cannot be installed directly, so you will have to create a virtual environment. We are using virtual environments to make the installation of packages and retention of correct versions as simple as possible. For this assignment we are going to use Miniconda + Python 3.7.

The instructions below are for DICE. You are free to use your own machine, but we cannot offer support for non-DICE machines. Similarly, this installation is for CPUs and we cannot offer support for GPU programming or any attempts to execute the assignment on a cluster such as `mlp.inf.ed.ac.uk`. The package specification file (`requirements.txt`) will only work for Linux based systems, if you choose to use a different machine then see the note below.

Open a terminal on a DICE machine and follow these instructions. We are expecting you to enter these commands in one-by-one. Waiting for each command to complete will help catch any unexpected warnings and errors. The total installation is about 4.33GB, please ensure you have sufficient space using the `freespace` command on DICE.

First install Miniconda from the home directory of your DICE user space. *You can skip this stage if you already have Miniconda installed.*

```
$> wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
$> bash ./Miniconda3-latest-Linux-x86_64.sh
$> rm ./Miniconda3-latest-Linux-x86_64.sh
$> source ~/.bashrc
```

Now your default Python version should be 3.7. Confirm with `python3 -version`. Having ensured you have a right version of Python, proceed to create a new environment called `nlu`. We will use the file `requirements.txt` included in the assignment code to generate this environment.

1. Create a new folder for the assignment:
   `$> mkdir nlu_cw1`
   `$> cd nlu_cw1`
2. Copy all code and data from Learn into this folder and unzip the compressed file as required.
3. Create an environment using the provided `requirements.txt` file:
   `conda create -n nlu --file ./requirements.txt -c pytorch`
4. Activate the `nlu` virtual environment:
   `$> conda activate nlu`
5. **Optional** Clean your workspace to free up space:
   `$> conda clean -all`

You should now have all the required packages installed. You only need to create the virtual environment and perform the package installations (step 1-4) **once**. However, make sure you activate your virtual environment (step 2) **every time** you open a new terminal to work on your assignment. Remember to use the `conda deactivate` command to disable the virtual environment when you don't need it.

1. Activate the environment:
   `$> conda activate nlu`
2. Deactivate the environment (if you want to work on something else):
   `$> conda deactivate nlu`

Using a different machine The above installation is designed for usage on DICE, but will also work for other Linux machines. If you are using Mac or Windows, you will have difficulty installing the required packages using the `requirements.txt` file as they are not designed for your computer. You can replace commands 3 & 4 using the following commands:

1. `$> conda create -n nlu python=3.7.5`
2. `$> conda activate nlu`
3. `$> conda install tqdm=4.40.2 numpy=1.18.5 gensim=3.8.0 pandas=0.25.3 seaborn=0.9.0 matplotlib=3.1.1`

4. `$> conda install pytorch=1.3.1 torchvision=0.4.2 cpuonly=1.0 -c pytorch`

Terminology/definitions Most neural network components, as well as their architecture and functionality, can be described using *matrix* and *vector* mathematical operations. Matrix and vector notation in the literature is inconsistent, so for this assignment we will use the following conventions:

(1) *Matrices* are assigned bold capital letters, e.g., $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$.

(2) *Vectors* are written in bold lower-cased letters, e.g., $\mathbf{x}$, $\mathbf{s}$, $\mathbf{net}_{in}$, $\mathbf{net}_{out}$.

(3) For a matrix $\mathbf{M}$ and a vector $\mathbf{v}$, $\mathbf{Mv}$ represents their *matrix-vector (dot) product*.[2]

(4) For vectors $\mathbf{v}$ and $\mathbf{w}$ of equal length n, $\mathbf{v} \circ \mathbf{w}$ represents their *element-wise product*:

$$\mathbf{v} \circ \mathbf{w} = [\mathrm{v}_0\mathrm{w}_0, \mathrm{v}_1\mathrm{w}_1, \ldots, \mathrm{v}_n\mathrm{w}_n]$$

Similarly, $\mathbf{v} + \mathbf{w}$ and $\mathbf{v} - \mathbf{w}$ express element-wise addition and subtraction.

(5) For vectors $\mathbf{v}$ and $\mathbf{w}$, $\mathbf{v} \otimes \mathbf{w}$ represents their *outer product*.[3]

(6) In recurrent neural networks, we process a *sequence* (or *time*), where each component is in a different state depending on the position in the sequence (or time step). We use the notation $\mathbf{M}^{(t)}$, $\mathbf{v}^{(t)}$ to refer to matrices and vectors at time step t.

Provided code and use of NumPy We provide a number of template files which you must use to write your code. `rnn.py` and `gru.py` are the implementation of the models, `runner.py` is used to train the models and define loss functions. We also provide an additional module, `rnnmath.py`, which consists of helper functions you can use. Finally, we provide `test.py`, which performs a very basic test of your code. Please familiarize yourself with the provided code and make sure you **don't** change the provided function signatures. Other classes should not be changed.

Throughout this assignment, you are required to use NumPy methods for all matrix/vector operations. **Do not try to implement matrix/vector functionality on your own.** If you need help with NumPy, please refer to its documentation[4] and look for answers on Google before asking on EdStem.

[2] https://en.wikipedia.org/wiki/Matrix_multiplication
[3] https://en.wikipedia.org/wiki/Outer_product
[4] http://docs.scipy.org/doc/numpy/reference/index.html

Introduction

For this assignment, you are asked to implement some basic functionality of a Recurrent Neural Network (RNN) for Language Modeling (LM). Given a word sequence $w_1, w_2, \ldots, w_t$, a language model predicts the next word w_{t+1} by modeling:

$$P(w_{t+1} \mid w_1, \ldots, w_t).$$

Below, you will be introduced to the main elements of a simple RNN for LM, based on the model proposed by Mikolov *et al.* (2010). In Question 2, you will implement its core word prediction functionality and its training by implementing a loss function and the model's gradient accumulation through backpropagation. In Question 3, you will adapt the model to the agreement prediction task as well as implementing a GRU (gate recurrent unit) cell Cho *et al.* (2014). Using this code you will compare the ability of RNNs and GRUs to learn long range dependencies.

Extra Resources:

- For understanding RNNs and BPTT please read Guo (2013). It is a fairly formal treatment but it explains everything in detail and defines all the notation.
- For understanding LSTMs I recommend Chapter 4 of Alex Graves thesis Graves (2012).
- This tutorial is also useful Chistopher (2015).

Recurrent Neural Networks

A recurrent neural network for language modeling uses feedback information in the hidden layer to model the "history" $w_1, w_2, \ldots, w_t$ in order to predict w_{t+1}. Formally, at each time step, the model needs to compute:

$$\begin{aligned}
\mathbf{s}^{(t)} &= f\left(\mathbf{net}_{in}^{(t)}\right) && (1)\\
\mathbf{net}_{in}^{(t)} &= \mathbf{Vx}^{(t)} + \mathbf{Us}^{(t-1)} && (2)\\
\hat{\mathbf{y}}^{(t)} &= g\left(\mathbf{net}_{out}^{(t)}\right) && (3)\\
\mathbf{net}_{out}^{(t)} &= \mathbf{Ws}^{(t)} && (4)
\end{aligned}$$

where $f()$ and $g()$ are the *sigmoid* and *softmax* activation functions respectively, $\mathbf{x}^{(t)}$ is the one-hot vector representing the vocabulary index of the word w_t, $\mathbf{net}_{in}^{(t)}$ and $\mathbf{net}_{out}^{(t)}$

are the activations for the hidden and output layers, and $\mathbf{s}^{(t)}$ and $\hat{\mathbf{y}}^{(t)}$ are the corresponding hidden and output vectors produced after applying the *sigmoid* and *softmax* nonlinearities.

For a given input $[w_1, w_2, \ldots, w_t]$, the probability of the next word at time step $t+1$ can be read from the output vector $\hat{\mathbf{y}}^{(t)}$:

$$P(w_{t+1} = j \mid w_t, \ldots, w_1) = \hat{y}_j^{(t)} \tag{5}$$

The parameters to be learned are:

$$\mathbf{U} \in \mathbb{R}^{D_h \times D_h} \quad \mathbf{V} \in \mathbb{R}^{D_h \times |V|} \quad \mathbf{W} \in \mathbb{R}^{|V| \times D_h} \tag{6}$$

where $\mathbf{U}$ is the matrix for the recurrent hidden layer, $\mathbf{V}$ is the input word representation matrix, $\mathbf{W}$ is the output word representation matrix, and D_h is the dimensionality of the hidden layer.

Question 1: Training RNNs [30 marks]

When training RNNs, we need to propagate the errors observed at the output layer $\hat{\mathbf{y}}$ back through the network, and adjust the weight matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ to minimize the observed loss w.r.t. a desired output. There are several loss functions suitable for use in RNNs. In RNN language models, an effective loss function is the cross-entropy loss:

$$J^{(t)}(\theta) = -\sum_{j=1}^{|V|} d_j^{(t)} \log \hat{y}_j^{(t)} \tag{7}$$

where $\mathbf{d}^{(t)} = [d_1^{(t)}, d_2^{(t)}, \ldots, d_{|V|}^{(t)}]$ is the one-hot vector representing the vocabulary index of desired output word at time t. In order to evaluate the model's performance, we average the cross-entropy loss across all steps in a sentence and across all sentences in the dataset.

(a) In the file `rnn.py`, implement the method `predict` of the `RNN` class. The method is used for *forward prediction* in your RNN and takes as input a sentence as a list of word indices $[w_1, \cdots, w_n]$. The return values are the matrices produced by concatenating hidden vectors $\mathbf{s}^{(t)}$ and output vectors $\hat{\mathbf{y}}^{(t)}$ for $t = 1, 2, \ldots, n$.[5] **[5 marks]**

[5] See the provided documentation on `rnn.py` for more details on the functions you need to implement.

(b) In `runner.py`, implement the methods `compute_loss` and `compute_mean_loss`. Given a sequence of input words $w = [w_1, \ldots, w_n]$ and a sequence of desired output words $d = [d_1, \ldots, d_n]$, `compute_loss` should return the total loss produced by the model's predictions for the sentence. The `compute_mean_loss` should compute the average loss over a corpus of input sentences. It should average across all words in all sentences of the given corpus. **[5 marks]**

Optimizing the loss using backpropagation means we have to calculate the update values Δ w.r.t. the gradients of our loss function for the observed errors. For the output layer weights, at time step t we accumulate the matrix $\mathbf{W}$ updates using:

$$\Delta\mathbf{W} = \eta \sum_{p=1}^{n} \delta_{out,p}^{(t)} \otimes \mathbf{s}_p^{(t)} \tag{8}$$

$$\delta_{out,p}^{(t)} = (\mathbf{d}_p^{(t)} - \hat{\mathbf{y}}_p^{(t)}) \circ g'(\mathbf{net}_{out,p}^{(t)}) \tag{9}$$

where η is the learning rate and p indicates the index of the current training pattern (sentence). We then further propagate the error observed at the output back to $\mathbf{V}$ with:

$$\Delta\mathbf{V} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t)} \otimes \mathbf{x}_p^{(t)} \tag{10}$$

$$\delta_{in,p}^{(t)} = \mathbf{W}^T \delta_{out,p}^{(t)} \circ f'(\mathbf{net}_{in,p}^{(t)}) \tag{11}$$

The derivatives of the softmax and sigmoid functions are respectively given as[6]:

$$g'(\mathbf{net}_{out,p}^{(t)}) = \vec{1} \tag{12}$$

$$f'(\mathbf{net}_{in,p}^{(t)}) = \mathbf{s}_p^{(t)} \circ (\vec{1} - \mathbf{s}_p^{(t)}) \tag{13}$$

Finally, in order to update the recurrent weights $\mathbf{U}$, we need to look back one step in time:

$$\Delta\mathbf{U} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t)} \otimes \mathbf{s}_p^{(t-1)} \tag{14}$$

(c) In `rnn.py`, implement the method `acc_deltas` that accumulates the weight updates for $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ for a *truncated* backpropagation through the RNN, where we only look back one step in time as described above, rather than the entire computation graph. **[10 marks]**

[6] We use $\vec{1}$ as shorthand for the all-ones vector of appropriate length.

Now we have implemented truncated backpropagation (BP) for recurrent networks—that is, RNNs that just look at the previous hidden layer when accumulating $\Delta\mathbf{U}$ and $\Delta\mathbf{V}$. We can extend truncated backpropagation to look at the previous τ time steps during backpropagation. At time t, the updates $\Delta\mathbf{W}$ can be derived as before. For $\Delta\mathbf{U}$ and $\Delta\mathbf{V}$, we additionally recursively update at times $(t-1)$, $(t-2)$ $\cdots(t-\tau)$:

$$\Delta\mathbf{V} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t-\tau)} \otimes \mathbf{x}_p^{(t-\tau)} \tag{15}$$

$$\Delta\mathbf{U} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t-\tau)} \otimes \mathbf{s}_p^{(t-\tau-1)} \tag{16}$$

$$\delta_{in,p}^{(t-\tau)} = \mathbf{U}^T \delta_{in,p}^{(t-\tau+1)} \circ f'(\mathbf{net}_{in,p}^{(t-\tau)}) \tag{17}$$

(d) Implement the method `acc_deltas_bptt` that accumulates the weight updates for **U**, **V** and **W** using backpropagation through time for τ time steps. **[10 marks]**

There's one last thing you'll need to do before your models will train: you'll need to complete the implementation of the `train-lm-rnn` mode in the `__main__` method of the code in `runner.py`. For this, you will minimally need to instantiate the RNN class, instatiate the Runner class with the RNN and call the `train` method on the RNN with the appropriate arguments, and save the resulting matrices. You may find it useful for your own understanding to log different aspects of the training process here.

You do not need to report anything in your writeup for Question 1. It will be evaluated solely on the basis of your code.

Question 2: Language Modeling [15 marks]

By now you should have everything in place to train a full Recurrent Neural Network using backpropagation through time. In the following questions, we will use the training and development data provided in `wiki-train.txt` and `wiki-dev.txt`. The training data consists of sentences from the parsed English Wikipedia corpus from Linzen *et al.* (2016), and each input/output pair x, d is of the form $[w_1, \cdots w_n]$ / $[w_2, \cdots w_{n+1}]$ that is, the desired output is always the next word of the current input:

time index	t=1	t=2	t=3	t=4
input:	Banks	struggled	with	the
output:	struggled	with	the	crisis

The `utils.py` module provides functions to read the Wikipedia data, and the `__main__` method of the `runner.py` module provides some starter code for training your models.

Use mode `train-lm-rnn` to train your language model. In order to start the training code you first have to instantiate the RNN and then a runner in order to call the `train` method.

(a) Perform parameter tuning using a subset of the training and development sets. You must use a fixed vocabulary of size $2,000$, and consider all combinations of the following parameter settings:

learning rate: 0.5, 0.1, or 0.05 number of hidden units: 25 or 50
number of steps to look back in truncated backpropagation: 0, 2, or 5

The mode `train-lm-rnn` in `runner.py` allows for more parameters, which you are free to explore. You should tune your model to maximize generalization performance (minimize cross-entropy loss) on the dev set. For these experiments, use the first $1,000$ sentences of both the training and development sets and train for 10 epochs.[7] Report your findings **and interpret them**. Your interpretation need not be more than a paragraph. **[10 marks]**

(b) Using your best parameter settings found in (a), train an RNN on a much larger training set. Use a fixed vocabulary size of 2000, train on 25,000 sentences, and, as before, use the first $1,000$ development sentences to evaluate the model's performance during training. When your model is trained[8], evaluate it on the **test** set and report the mean loss, as well as the perplexity your model achieves. Save your final learned matrices **U**, **V** and **W** as files `rnn.U.npy`, `rnn.V.npy` and `rnn.W.npy`, respectively. **[5 marks]**

Question 3: Predicting Agreement with RNNs and GRUs [20 marks]

The form of an English third-person present tense <u>verb</u> depends on whether the **head** of the syntactic subject is plural or singular. For example, native English speakers strongly prefer sentences (i) and (iv) below, and regard (ii) and (iii) as ungrammatical, as indicated by the *:

i) The **key** <u>is</u> on the table.

ii) *The **key** <u>are</u> on the table.

iii) *The **keys** <u>is</u> on the table.

[7] Note that training models might take some time. For example, a sweep of the parameters settings described above should take roughly 2 hours on a student lab DICE machine. Please avoid using `student.compute` to train your models as run times will become very slow on a busy server.

[8] This should also take roughly 2 hours. Again, avoid using `student.compute`.

iv) The **keys** <u>are</u> on the table.

This agreement tends to persist even when the head of the subject is separated from the verb by intervening words:

v) The **keys** to the cabinet <u>are</u> on the table.

Agreement rules like this occur in many languages, and are often more complex than in English. Our goal for this question will be to test (in a limited way) whether an RNN can learn them. For our first test, we will train a model predict agreement using direct supervision. That is, we will give our model the sequence of words preceding the verb, and we will ask it to predict whether the verb is singular (VBZ), or plural (VBP). Our training and test data will be in this form:

time index	t=1	t=2	t=3	t=4	t=5
input:	The	keys	to	the	cabinet
output:					VBP

Since the task is now binary classification, we must make some changes to the RNN. Instead of making predictions at **every** time step, we only make a prediction at the **final** time step.

(a) Implement new functions for weight updates (`acc_deltas_np`, `acc_deltas_bptt_np`, rnn.py), loss fuction (`compute_loss_np`, runner.py), and prediction accuracy (`compute_acc_np`, runner.py) to reflect the structure of the number prediction problem. **[5 marks]**

Since the head of the subject may be arbitrarily far from the verb, this problem is a natural application of RNNs. The model needs to learn long-term dependencies to correctly predict whether a verb is singular or plural.

Gated Recurrent Unit

The Gated Recurrent Unit (GRU) is special type of RNN which is capable of learning long range dependencies. The overall structure is the same but each RNN cell is replaced by a GRU cell. Formally, at each time step the model computes:

$$
\begin{aligned}
\mathbf{r}^{(t)} &= f\left(\mathbf{V}_r\mathbf{x}^{(t)} + \mathbf{U}_r\mathbf{s}^{(t-1)}\right) && (18)\\
\mathbf{z}^{(t)} &= f\left(\mathbf{V}_z\mathbf{x}^{(t)} + \mathbf{U}_z\mathbf{s}^{(t-1)}\right) && (19)\\
\tilde{\mathbf{h}}^{(t)} &= tanh\left(\mathbf{V}_h\mathbf{x}^{(t)} + \mathbf{U}_h\left(\mathbf{r}^{(t)} \circ \mathbf{s}^{(t-1)}\right)\right) && (20)\\
\mathbf{s}^{(t)} &= \mathbf{z}^{(t)} \circ \mathbf{s}^{(t-1)} + \left(1 - \mathbf{z}^{(t)}\right) \circ \tilde{\mathbf{h}}^{(t)} && (21)\\
\mathbf{net}_{out}^{(t)} &= \mathbf{W}\mathbf{s}^{(t)} && (22)\\
\hat{\mathbf{y}}^{(t)} &= g\left(\mathbf{net}_{out}^{(t)}\right) && (23)
\end{aligned}
$$

where $f()$, $g()$ and $tanh()$ are the *sigmoid*, *softmax* and *tanh* activation functions respectively. $\mathbf{x}^{(t)}$ is the one-hot vector representing the vocabulary index of the word w_t. $\mathbf{r}_{in}^{(t)}$ and $\mathbf{z}_{in}^{(t)}$ are the reset and update gates respectively. $\tilde{\mathbf{h}}^{(t)}$ is an additional "candidate" hidden state that uses the reset gate to remove irrelevant information. $\mathbf{s}^{(t)}$ is the actual hidden state which is obtained using the update gate and $\tilde{\mathbf{h}}^{(t)}$. $\mathbf{net}_{out}^{(t)}$ is the activation of the output layer.

The parameters to be learned are:

$$
\begin{aligned}
\mathbf{U}_r \in \mathbb{R}^{D_h \times D_h} \quad \mathbf{V}_r \in \mathbb{R}^{D_h \times |V|} && (24)\\
\mathbf{U}_z \in \mathbb{R}^{D_h \times D_h} \quad \mathbf{V}_z \in \mathbb{R}^{D_h \times |V|} && (25)\\
\mathbf{U}_h \in \mathbb{R}^{D_h \times D_h} \quad \mathbf{V}_h \in \mathbb{R}^{D_h \times |V|} && (26)\\
\mathbf{W} \in \mathbb{R}^{|V| \times D_h} && (27)
\end{aligned}
$$

where $\mathbf{U}$ is the matrix for the recurrent hidden layer, $\mathbf{V}$ is the input word representation matrix, $\mathbf{W}$ is the output word representation matrix, and D_h is the dimensionality of the hidden layer. Unlike the RNN, this implementation of a GRU learns three separate sets of U and V parameters denoted by r, z and h. These subscripts represent the reset, update and hidden state parameters respectively.

(b) In the file `gru.py`, implement the `forward` method. Note this method only needs to perform the forward prediction for a single time step, the prediction for an entire sequence is already handled by `gru_abstract.py` if the forward method is correctly implemented. It takes as input the single word index for the current time step w_t and the hidden state vector from the previous time step $\mathbf{s}^{(t-1)}$. The return values are the output vector $\mathbf{y}^{(t)}$, the reset gate vector $\mathbf{r}^{(t)}$, the update gate vector $\mathbf{z}^{(t)}$, the "candidate" hidden state vector $\tilde{\mathbf{h}}^{(t)}$ and the new hidden state vector $\mathbf{s}^{(t)}$. **[5 marks]**

(c) Next, implement the `acc_deltas_np` and `acc_deltas_bptt_np` methods in the `gru.py` class. The `gru.py` class provides a `self.backward()` method that takes $\delta^{(t)}_{out,p}$ given in Equation 9 as an input and performs the backpropagation for the rest of the GRU model. Specifically, `self.backward()` takes as input the list of word indicies x, the current time step t, the gradients of the output $\delta^{(t)}_{out,p}$ and an optional parameter *steps*. *steps* defines the number of steps for BPTT and is set to 0 by default. As for the RNN `acc_deltas_np` and `acc_deltas_bptt_np` should reflect the structure of the number prediction problem. **[5 marks]**

You'll need to complete the implementation of the `train-np-rnn` and `train-np-gru` modes in the `__main__` method of the `runner.py` file. Check your implementation of the RNN by running the code:

```
$> python runner.py train-np-rnn data_dir hdim lookback learning_rate
```

and the GRU by running:

```
$> python runner.py train-np-gru data_dir hdim lookback learning_rate
```

(d) Using the first 2,000 training sentences, and a learning rate of 0.5 training, train your models for 10 epochs and compare the accuracy of the RNN and the GRU models by performing a sweep of the number of hidden units, setting them to: 10, 25 or 50.

Report your the results for both models and briefly describe how they differ from each other. **[5 marks]**

Question 4: Comparing Recurrent Models [15 marks]

This is a more open-ended question that gives you considerable flexibility in how to approach it. We will mark your answer in terms of how plausible your hypothesis is and how good the evidence is (argumentation and results) that you provide to support that hypothesis.

In Question 3, you tested GRUs and RNNs on the agreement task. However, you did not use backpropagation through time (the number of BPTT steps was set to 0). The aim of this question is to compare the two models when backpropagation through time is used. Specifically, you should compare how the two architectures perform on the number prediction task as number of BPTT steps increases .

To answer this question, you need to go through the following steps:

- Come up with a **hypothesis** about the behaviour you expect to see from the two models. Describe your hypothesis, explain how you came up with it, and why you think it is plausible.

- Run a set of **experiments** with the two models. Explain why these experiments are suitable for testing your hypothesis and describe how you ran them.
- Present the **results** of your experiments (typically as a graph or in a table) and provide an **interpretation** of the results in the context of your hypothesis. Do they support it or contradict it? Why?

To carry out your experiments, it may be necessary, for example, to increase the number of training samples or change the number of epochs. Additionally, you may need to change the runner class to log additional information during training; in this case we recommend doing this in a new runner class to ensure that any changes to the code do not break the automatic marking.

Please note that the code will not be marked for this question; we will mark only the writeup, as outlined in the bullet points above. **[15 marks]**

References

Olah Chistopher. Understanding LSTM Networks. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015. Unpubl. Blog.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. PhD thesis, Technische Universitat Munchen, 2012.

Kristina Gulordava, Piotr Bojanowski, Edouard Grave, Tal Linzen, and Marco Baroni. Colorless green recurrent networks dream hierarchically. In *Proceedings of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2018.

Jiang Guo. Backpropagation Through Time. `http://ir.hit.edu.cn/~jguo/docs/notes/bptt.pdf`, 2013. Unpubl. ms., Harbin Institute of Technology.

Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535, 2016.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, volume 2, page 3, 2010.

Acknowledgements

This assignment is based in parts on code and text kindly provided by Richard Socher, from the course on “Deep Learning for Natural Language Processing”.[9]

[9] http://cs224d.stanford.edu/